NAME

hoc — (high-order calculator) [interactive floating-point language]

SYNOPSIS

hoc [**−author**] [**−copyright**] [**−D***name*] [**−D***name*=*number*] [**−D***name*=*"string"*] [**−D***name*=*symbol*] [**−?**] [**−help**] [**−no-banner**] [**−no-cd**] [**−no-help-file**] [**−no-load**] [**−no-logfile**] [**−no-readline**] [**−no-save**] [**−no-site-file**] [**−no-translation-file**] [**−no-user-file**] [**−quick**] [**−secure**] [**−silent**] [**−trace-file-opening**] [**−U***name*] [**−version**] [*file* ...]

hoc can be built in up to three floating-point precisions, corresponding to the C/C++ data types *float*, *double*, and *long double*. These are conventionally distinguished by suffixes indicating the number of bits in the floating-point system: **hoc32**, **hoc64** (same as **hoc**), **hoc80**, and **hoc128**. Support for precisions other than 64-bit *double* is deficient, or nonexistent, in C/C++ implementations on several operating systems, so some **hoc** precisions may be unavailable on your system.

OPTIONS

hoc options can be prefixed with either one or two hyphens, and can be abbreviated to any unique prefix. Thus, **−a**, **−author**, and **−−auth** are equivalent.

To avoid confusion with options, if a filename begins with a hyphen, it must be disguised by a leading absolute or relative directory path, e.g., */tmp/-foo.hoc* or *./-foo.hoc*.

−author
Display an author credit, and software distribution information, on the standard error unit, *stderr*, and then terminate with a success return code. Sometimes an executable program is separated from its documentation and source code; this option provides a way to recover from that.

−copyright
Display copyright information on the standard error unit, *stderr*, and then terminate with a success return code.

−D*name*
Define the numeric variable *name* to have the value 1.

−D*name*=*number*
Define the numeric variable *name* to have the value *number*.

If = is changed to :=, the assignment is permanent: *name* cannot then subsequently be redefined.

−D*name*=*"string"*
Define the string variable *name* to have the value *"string"*.

If = is changed to :=, the assignment is permanent: *name* cannot then subsequently be redefined.

Since command shells on some operating systems interpret quotation marks, it is usually necessary to protect them. On UNIX-like systems, use **'−D***name*=*"string"***'** or if the value contains no characters that are significant to the command shell, **−D***name*=*"string**"*

−D*name*=*symbol*
Define the numeric or string variable *name* to have the value of that of *symbol*, which must be an existing named constant or variable.

If = is changed to :=, the assignment is permanent: *name* cannot then subsequently be redefined.

−help or **−?**
Display a help message on *stderr*, giving a brief usage description, and then terminate with a success return code.

−no-banner
Suppress any welcome banners normally printed by dynamically-loaded library code.

This option can also be set via the **hoc** system variable **__BANNER__**, but it must be set in an initialization file before code in that file to print the welcome banner is reached.

−no-cd
Disable the change-directory function, **cd(***dir***)**, and the print-working-directory function, **pwd( )**.

This option is a security feature: it takes effect only *after* all initialization files have been processed.

–no-help-file Suppress loading of system-wide **hoc** help files at startup.

–no-load Disable the **load( )** function. It will continue to be recognized, but when invoked, will simply print a warning that it has been disabled.

This option is a security feature: it takes effect only *after* all initialization files have been processed.

–no-logfile Disable the **logfile( )**, **logon( )**, and **logoff( )** functions. They will continue to be recognized, but when invoked, will simply print a warning that they have been disabled.

This option is a security feature: it takes effect only *after* all initialization files have been processed.

–no-readline Suppress use of the GNU *readline* library: command completion, editing and recall are then not available.

On some systems, it may be necessary to use this option when **hoc** is used in international mode (see the **INTERNATIONALIZATION** section below) in order to get accented letters displayed properly.

–no-save Disable the **save( )** function. It will continue to be recognized, but when invoked, will simply print a warning that it has been disabled.

This option is a security feature: it takes effect only *after* all initialization files have been processed.

–no-site-file Suppress loading of the system-wide non-help startup files.

–no-translation-file Suppress loading of the system-wide message translation files.

–no-user-file Suppress loading of the user-specific startup file.

–quick Suppress loading of all startup files: this option is equivalent to **–no-help-file –no-site-file –no-translation-file –no-user-file**.

–secure Enable all security features: this option is equivalent to **–no-cd –no-load –no-logfile –no-save**, and in addition, makes it impossible to trace file openings by setting the **__DEBUG_OPEN__** system variable to a nonzero value.

–silent Suppress printing of prompts for interactive input. **hoc** never prompts when it is reading noninteractive files.

The **hoc** system variable **__VERBOSE__** can also be set to zero at run time to turn off prompts; setting it to nonzero turns them back on.

The **hoc** system variable **__PROMPT__** contains the prompt string: it can be redefined at any time.

–trace-file-opening Set the **hoc** variable **__DEBUG_OPEN__** to a nonzero value to trace input file opening attempts.

–U*name* Undefine the variable *name*.

–version Display the program version number and release date on *stderr*, and then terminate with a success return code.

DESCRIPTION

hoc interprets a simple language for floating-point arithmetic, at about the level of Basic, with C-like syntax and functions. However, unlike Basic, **hoc** has particularly rich support for floating-point arithmetic, and its facilities are certainly better than that standardly provided by most programming languages, such as C, C++, and Fortran.

hoc recognizes the three popular line-ending conventions in text files: line feed (LF) (UNIX), carriage return (CR) (Apple MacOS), and CR LF (PC DOS, Microsoft Windows, and several older systems). It also ignores one or more Ctl-Z characters at end of file, a horrid relic of some legacy desktop operating systems. Thus, its input files should not even require line-terminator translation when they are moved between systems.

To get a flavor of what typical **hoc** code looks like, visit the `*.hoc` and `*.rc` files in the **hoc** installation directory tree: see the **INITIALIZATION FILES** section below for their location.

The named *file*s are read and interpreted in order. If no *file* is given or if *file* is −, **hoc** interprets the standard input.

See the **INPUT FILE SEARCH PATH** section below for details on how **hoc** finds input files.

hoc input consists of *expressions* and *statements*. Expressions are evaluated and their results printed. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call **print**.

Word completion

When **hoc** has been built with the GNU *readline* library, word completion can be used to save typing effort. It is normally requested by an ESCape character following a prefix of a word in **hoc**'s symbol table: **hoc** will respond with an audible beep, and a list of words that match that prefix, or if only one word matches, it will silently complete the word:

```
hoc> c<ESCape>
cbrt ceil copysign cos cosd cosh
hoc> co<ESCape>
copysign cos cosd cosh
hoc> cop<ESCape>
hoc> copysign
```

The character used to request completion can be changed: see the **INITIALIZATION FILES** section below.

Command history and editing

When **hoc** has been built with the GNU *readline* library, convenient command history and editing support is available, much like it is in the UNIX **bash**(1), **ksh**(1) and **tcsh**(1) shells, and in a few GNU programs, like the **bc**(1) and **genius**(1) calculators. The default history and editing mode is **emacs**(1)-style; you can also get **vi**(1)-style by suitable customizations: see the **INITIALIZATION FILES** section below.

In the default mode, *C-p* (hold the Control key down while typing p) moves up in the history list, *C-n* moves down, *C-b* moves backward in the current line, *C-f* moves forward, *C-d* deletes forward, *DELete* deletes backward, *C-a* moves to the beginning of the line, *C-e* moves to the end of the line, *C-l* repaints the screen, reprinting the current line at the top, and *RETurn* resubmits the line for execution.

For more details, consult the *GNU Readline Library* manual, available online in the *info* system. In **emacs**(1); type *C-h i mreadline* to get there.

Numbers

All numbers in **hoc** are stored as double-precision floating-point values.

On systems with IEEE 754 arithmetic, such numbers are capable of representing integers of up to 53 bits exactly, excluding the sign bit. This is an integer range of −(2ˆ53) . . . 2ˆ53, or −9,007,199,254,740,992 . . . 9,007,199,254,740,992.

Some systems may have **hoc** variants named **hoc32**, **hoc64**, **hoc80**, or **hoc128**, in which case, **hoc**'s default precision is indicated by the program name: 32-bit, 64-bit, 80-bit, or 128-bit. **hoc** and **hoc64** are identical on all systems with IEEE 754 arithmetic.

For **hoc32**, on systems with IEEE 754 arithmetic, numbers are capable of representing integers up to 24 bits exactly, excluding the sign bit. This is an integer range of −(2ˆ24) . . . 2ˆ24, or −16,777,216 . . . +16,777,216.

For **hoc80**, on systems with IEEE 754 arithmetic, numbers are capable of representing integers up to 64 bits exactly, excluding the sign bit. This is an integer range of −(2ˆ64) . . . 2ˆ64, or −18,446,744,073,709,551,616 . . . +18,446,744,073,709,551,616.

For **hoc128**, on systems with IEEE 754 arithmetic, numbers are capable of representing integers up to 112 bits exactly, excluding the sign bit. This is an integer range of −(2^112) ... 2^112, or −5,192,296,858,534,827,628,530,496,329,220,096 ... +5,192,296,858,534,827,628,530,496,329,220,096.

Numbers may be signed, and may optionally contain a decimal point, a power-of-ten exponent, and a precision suffix. The exponent consists of an exponent letter, one of *d*, *D*, *e*, *E*, *q*, or *Q* (supported by one or more of Ada, C, C++, Fortran, Java, and Pascal), followed by an optionally-signed integer. The precision suffix (used by C, C++, and Java) is one of *f*, *F*, *l*, or *L*. The suffix does *not* affect the precision of the constant for **hoc**: it is recognized only to simplify incorporation into **hoc** programs of numbers from programs in other languages, and their output.

A hexadecimal floating-point number format, introduced in the latest ISO C Standard, *ISO/IEC 9899:1999 (E) Programming languages — C*, usually known by its short name, *C99*, is also supported, and implemented by portable private code in **hoc**. This format consists of an optional sign, then *0x* or *0X*, followed by one or more hexadecimal digits (*0–9 A–F a–f*) containing at most one hexadecimal point, optionally followed by a binary (power-of-two) exponent consisting of *p* or *P* followed by an optionally-signed decimal integer. If present, the exponent may optionally be followed by a precision suffix letter: one of *f*, *F*, *l*, or *L*. Thus, *−0x1.00000p8*, *−0x100*, *−0x100000p-12f*, *−0x10p+4L*, *−0x1p+8*, *−0x1p00008*, and *−0x1p8* all represent the decimal number *−256*.

The hexadecimal format, while awkward for humans, has the advantage of guaranteeing exact input/output conversions on all platforms, and **hoc** consequently uses this format in files created by the **save( )** command.

Strings

String constants are delimited by quotation marks (`"..."`), and may not span multiple lines, unless the embedded line breaks are each prefixed with a backslash, which is removed, leaving the newline in the string.

All characters in 1 ... 255 are representable in strings; as in C and C++, character 0 (ASCII NUL) is reserved as a string terminator.

In string constants, nonprintable characters may be represented by the usual escape sequences defined in Standard C and Standard C++, plus one extension (**\E**):

****	backslash: ASCII decimal 92.
\"	quotation mark: ASCII decimal 34.
\a	alert or bell (ASCII BEL: decimal 7).
\b	backspace (ASCII BS: decimal 8).
\E	escape (ASCII ESC: decimal 27).
\f	formfeed (ASCII FF or NP: decimal 12).
\n	newline (ASCII LF or NL: decimal 10).
\r	carriage return (ASCII CR: decimal 13).
\t	horizontal tab (ASCII HT: decimal 9).
\v	vertical tab (ASCII VT: decimal 11).
\o \oo \ooo	octal character number (**o** = *0–7*) in one to three digits.
\xh...	hexadecimal character (**h** = *0–9A–F* or *0–9a–f*) in one or more digits.

Backslash followed by any other character than those listed is simply discarded: **\W** reduces to **W**.

Variables

Variable names consist of an initial letter or underscore, followed by any number of letters, underscores, or digits. Lettercase is *significant*. Letters are considered to be *A–Z*, *a–z*, and any characters in the range 160 ... 255 of an 8-bit character set. Use of characters in the latter range is normally not recommended, because they are often difficult, or impossible, to generate on some computer keyboards. Nevertheless, it does permit non-English words to be spelled correctly; see the **INTERNATIONALIZATION** section below.

Underscore (_) by itself is a reserved variable containing the value of the last *numeric* expression evaluated. Double underscore (_ _) is a reserved variable containing the value of the last *string* expression evaluated. They cannot be assigned to by user code.

Predefined numeric constants and variables

Certain immutable named constants are already initialized:

CATALAN	Catalan's constant: sum((−1)^i/(2*i+1)^2, i = 0..infinity) = approximately 0.915965594177219015054603514932. . .
CLASS_xxx	One of eleven possible return values from the **class(x)** function; see the section **DESCRIPTIONS OF BUILT-IN FUNCTIONS AND PROCEDURES** below for details.
DEG	180/PI, degrees per radian
E	base of natural logarithms
GAMMA	Euler's constant: limit(sum(1/i,i=1. . .n) − ln(n), n → infinity) = approximately 0.577215664901532860606512090082. . .
INF or **Inf** or **Infinity**	IEEE-754 floating-point infinity
MAXNORMAL	Largest finite normalized floating-point number.
MINNORMAL	Smallest (in absolute value) nonzero normalized floating-point number.
MINSUBNORMAL	Smallest (in absolute value) subnormal floating-point number. If your computer system does not support subnormal numbers, this is identical to **MINNORMAL**.
NAN or **NaN**	IEEE-754 floating-point not-a-number
PHI	golden ratio: (1 + sqrt(5))/2 = approximately 1.61803398874989484820458683436. . .
PI	ratio of the circumference of a circle to its diameter, approximately 3.14159265358979323846264338327. . .
PREC	maximum number of significant digits in output, initially 17 on most systems (the precise value is computed dynamically, from Matula's 1968 result: *ceil(N/log_b(10) + 1)*, for a host floating-point system with *N* base-*b* digits). **PREC = 0** gives shortest 'exact' values.
QNAN or **QNaN**	IEEE-754 floating-point quiet not-a-number
SNAN or **SNaN**	IEEE-754 floating-point signaling not-a-number

More information on the floating-point constants is available in the **FLOATING-POINT ARITHMETIC** section below.

Predefined system constants and variables

hoc also provides a number of system constants and variables, adopting the C/C++ convention that names beginning with two underscores are reserved for the implementation:

_	[immutable number] Value of the last numeric expression printed (initialized to 0.0 on startup).
_ _	[immutable string] Value of the last string expression printed (initialized to an empty string on startup).
_ _BANNER_ _	[reassignable number] Nonzero (true) if printing of welcome banners is permitted. It can be changed by the **−no-banner** option.

__CPU_LIMIT__ [immutable number] Current limit on CPU use, in seconds. It is normally infinite, but can be reset by the **cpulimit( )** function.

__DATE__ [constant string] Date of the start of job execution, in the form `"Dec 16 2001"`. The day number has a leading space if only one digit is needed, so that the string always has constant width.

__FILE__ [constant string] Name of the current input file. This is `"/dev/stdin"` when **hoc** is reading from the standard input.

__FILE__[*n*] [constant string] Name of the *n*-th input file in the current job. This provides a history of exactly what files have been read. Because **hoc** does not yet support arrays, the only way to display these is with the **who( )** function.

__GID__ [constant number] Group numeric identifier code.

On operating systems that do not support the concept of group and user identifiers, it is set to zero.

__HOCRC__ [constant string] Pathless filename of the optional **hoc** user startup file; it is stored in the user's home directory.

__IEEE_754__ [constant number] Nonzero (true) if the host system supports IEEE 754 arithmetic.

__LINE__ [constant number] Number of the current input line in the file named by **__FILE__**.

__MAX_xxx__ [immutable number] One of several numeric constants that report current sizes of internal storage areas in **hoc** that grow as needed. See the **IMPLEMENTATION LIMITS** section below for details.

__PACKAGE_BUGREPORT__ [constant string] Where to report bugs.

__PACKAGE_DATE__ [constant string] Date of last modification of the software.

__PACKAGE_NAME__ [constant string] Program name.

__PACKAGE_STRING__ [constant string] Program name and version number.

__PACKAGE_VERSION__ [constant string] Program version number.

__PID__ [constant number] Numeric process identifier.

On operating systems that do not support such a concept, it is set to zero.

__PROMPT__ [reassignable string] Current prompt string. Prompting is controlled by the setting of **__VERBOSE__** (see below).

For example,

```
__PROMPT__ = "\n\E[7mInput:\E[0m "
```

will produce a blank line followed by a prompt in inverse video in terminal emulators, such as **xterm**(1) and DEC VT100, that follow the ANSI X3.64-1979 or ISO 6429-1983 terminal standards.

If **__PROMPT__** contains the two-character format string %d, that string will be replaced by the prompt count: for example, this silly setting

```
__PROMPT__="\E[4;5;34;43m[%d]\E[0m: "
```

will display the count digits in blue, and underlined, on a yellow background, in an **xterm**(1) window that supports text color attributes. [Run `dircolors -p` for more information on color settings.]

__READLINE__
: [constant number] Nonzero (true) if the GNU *readline* library is in use.

__SYSHOCDIR__
: [constant string] Name of the installation directory in which **hoc** startup files are stored.

__SYSHOCHLPBASE__
: [constant string] Pathless filename of the top-level startup help file.

__SYSHOCHLP__
: [constant string] Full filename of the top-level startup help file.

__SYSHOCPATH__
: [constant string] System directory search path that is substituted for an empty component in the **HOCPATH** environment variable input file directory search list.

__SYSHOCRCBASE__
: [constant string] Pathless filename of the top-level startup help file.

__SYSHOCRC__
: [constant string] Full filename of the top-level startup file.

__SYSHOCXLTBASE__
: [constant string] Pathless filename of the top-level translation file.

__SYSHOCXLT__
: [constant string] Full filename of the top-level translation file.

__TIME__
: [constant string] Local time-of-day (24-hour clock) of the start of job execution, in the usual hours, minutes, seconds form `"14:57:23"`.

__UID__
: [constant number] User numeric identifier code.

  On operating systems that do not support the concept of group and user identifiers, it is set to zero.

__VERBOSE__
: [reassignable number] Nonzero (true) if **hoc** should prompt for input from interactive files. The actual prompt string is controlled by the **__PROMPT__** variable.

  [NB: A bug in the GNU *readline* library (version 4.2a) makes this variable ineffective; it works correctly with the **–noreadline** option. The bug has been reported to the *readline* maintainers.]

Numeric expressions

Numeric expressions are formed with these C-like operators, listed by decreasing precedence.

`^`
: Exponentiation.

`! - ++ --`
: Logical negation, arithmetic negation, increment-by-one, decrement-by-one. As in C and C++, the latter two may be applied *before* a variable (acting first before taking the value), or *after* (taking the current value first, then acting).

`* / %`
: Multiply, divide, modulus.

`+ -`
: Add, subtract.

`> >= < <= <> == !=`
: Greater than, greater than or equal to, less than, less than or equal to, less than or greater than, equal to, not equal to.

The **<>** operator is *not* the same as **!=**; they differ when one of the operands is a NaN. Since NaNs are unordered, NaN <> NaN is 0 (false), while NaN != NaN is 1 (true).

&& Logical and. Both operands are *always* evaluated, unlike in C and C++, where the second is evaluated only if the first is nonzero (true).

|| Logical or. Both operands are *always* evaluated, unlike in C and C++, where the second is evaluated only if the first is zero (false).

= += -= *= /= %= := Assignment, assign the left-hand side the (sum, difference, product, dividend, or modulus) of its current value and the right-hand side, permanent assignment. The operator **:=** is a *one-time-only* assignment operator, used for defining permanent constants that cannot be redefined in the same **hoc** session.

As in C and C++, assignment is a right-associative expression whose value is the left-hand side. This means that `x = y = z = 3` is interpreted as `x = (y = (z = 3))`. That is, `3` is assigned to `z`, then that result is assigned to `y`, and finally, that result is assigned to `x`, so all three variables are assigned the value `3`. Similarly, `sqrt(x = 4)` assigns the value `4` to `x` before computing and returning its square root.

Expression lists in **print**-like statements, and in argument lists, are evaluated in strict *left-to-right* order. Thus, the output of expressions with side effects, such as

```
n = 3
print ++n, n++
```

is predictable: that example prints

```
4 4
```

String expressions

String expressions support only the relational operators (**> >= < <= == !=**) and the simple assignment operators (**= :=**), plus concatenation, which is indicated by two successive string expressions, without any specific operator, following the practice in C, C++, and **awk**(1). These two assignments are equivalent:

```
s = "hello"   ", "   "wor"   "ld"
s = "hello, world"
```

Numbers in string expressions are converted to strings according to the current precision variable, **PREC**.

```
k = 123
PREC = 4
s = "abc" k "def" PI
println s
abc123def3.142
```

Several string functions listed below augment string expressions.

Built-in functions and procedures

Longer documentation of the built-in functions and procedures is relegated to the later section, **DESCRIPTIONS OF BUILT-IN FUNCTIONS AND PROCEDURES**.

These numeric built-in functions take zero arguments: **rand**, **second**, **snan**, and **systime**.

These numeric built-in functions take one numeric argument: **abs**, **acos**, **acosh**, **asin**, **asinh**, **atan**, **atanh**, **cbrt**, **ceil**, **cos**, **cosd**, **cosh**, **double**, **erf**, **erfc**, **exp**, **expm1**, **exponent**, **factorial**, **floor**, **gamma**, **ilogb**, **int**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isqnan**, **issnan**, **issubnormal**, **J0**, **J1**, **lgamma**, **ln**, **log**, **log10**, **log1p**, **log2**, **macheps**, **nint**, **number**, **randl**, **rint**, **rsqrt**, **setrand**, **significand**, **sin**, **sind**, **single**, **sinh**, **sqrt**, **tan**, **tand**, **tanh**, **trunc**, **Y0**, and **Y1**.

These numeric built-in functions take two numeric arguments: **copysign**, **errbits**, **fmod**, **gcd**, **hypot**, **Jn**, **lcm**, **ldexp**, **logb**, **max**, **min**, **randint**, **nearest**, **nextafter**, **remainder**, **scalb**, and **Yn**.

These string built-in functions take zero arguments: **logoff**, **logon**, **now**, and **pwd**.

These string built-in functions take one argument: **cd**, **char**, **eval**, **ftoh**, **getenv**, **length**, **hexfp**, **hexint**, **htof**, **ichar**, **load**, **logfile**, **msg_translate**, **printenv**, **protect**, **set_locale**, **string**, **tolower**, **toupper**, and **who**.

These string built-in functions take two arguments: **index**, **putenv**, **save**, and **strftime**.

This string built-in function takes three arguments: **substr**.

These numeric functions take one symbol argument: **defined** and **delete**.

These symbol functions take one string argument: **symnum** and **symstr**.

These startup file procedures take no arguments: **author**, **dirs**, **help**, **help_xxx**, **news popd**, and **xd**.

The help system (described later) documents each of these functions, and any additional ones provided by startup files. Most have the same names as they do in C, C++, and Fortran, so many will already be familiar to users who have learned any of those programming languages.

Built-in functions and procedures are *immutable*: they cannot be redefined by the user in **hoc** code. User-defined variables, functions, and procedures can be redefined at any time to objects of the same type. Variables can be redefined to be functions or procedures. However, the reverse does not hold: once a name has been used as a function or procedure, it can only be redefined to be a new function or procedure.

The procedure **abort(message)** prints **message**, immediately terminates evaluation, and returns to the top-level interpreter, discarding and clearing the function/procedure call stack. It is equivalent to a similar internal function that **hoc** uses to recover from catastrophic errors. Use it sparingly!

The function **read(x)** reads a value into the variable **x**. The value must be either a number, or a quoted string, or an existing variable or named constant. The return value is 1 on success, or 0 on end-of-file; the function aborts for any other error condition.

The function **who(pattern)** produces a lengthy report of all of the named constants and variables with their current values, plus the names of all built-in functions and procedures, and all user-defined functions and procedures. Only those names which match the argument string, **pattern**, are included.

To print all symbols, use **who("*")**. The return value is always an empty string.

Symbols with three or more leading underscores are for internal use by **hoc**, and are thus considered *hidden*. They can only be shown by a suitable **pattern** argument to **who( )**. Hidden symbols are used for locale translations of embedded strings. See the **INTERNATIONALIZATION** section below for further details.

Statements

Control flow statements are **if–else**, **while**, and **for**, with braces for grouping.

The **break** statement exits from the body of a **for** or **while** loop, skipping evaluation of any post-body **for**-loop expression. Execution resumes with the statement that follows the loop body.

The **continue** statement exits from the current iteration of the body of a **for** or **while** loop. Execution resumes with evaluation of any post-body **for**-loop expression, and the conditional test that governs execution of the next iteration.

break and **continue** are illegal outside loop bodies.

Newline or semicolon ends a statement. Backslash-newline is equivalent to a space.

Functions and procedures are introduced by the words **func** and **proc**, followed by the function/procedure name, a parenthesized list of arguments, and the function/procedure body, which may be either a single statement, or a braced statement group.

The **return** statement is used to return a value from a function.

Variables inside the body are *local* by default: they are known only within the body, even if they have the same names as variables elsewhere.

Variables listed in a **global** statement are visible outside the body.

Arguments are passed by value, so it is impossible for the body to modify their values in the caller.

Here is an example to demonstrate these features:

```
proc show(x) \
{
    global last_x, last_xsq
    println x
    last_x = x
    last_xsq = x^2
    y = x
    x = 999999999
}
x = 5
show(x)
5
last_x
5
last_xsq
25
y
0
x
5
```

Built-in named constants are always globally visible, and thus need not be listed in a **global** statement; however, some people prefer to do so as a matter of documentation.

A **global** statement may appear outside a function/procedure body, but does nothing useful.

Inside a function/procedure body, a **global** statement affects only variables that follow it, so it should generally occur at the start. Here is what happens if you do not follow that recommendation:

```
y = 3
proc foo() \
{
    for (k = 1; k < 3; ++k) \
    {
        println y
        global y
        println y
    }
}
foo()
0
3
0
3
```

The first reference to `y` is to an uninitialized, and therefore zero by default, variable. The second one is to the global value. This behavior is a feature, not a bug, but you probably should not exploit it!

In older versions of **hoc**, function/procedure statement argument lists were empty, and within the body, numeric arguments were referred to as **$1**, **$2**, etc., and string arguments as **$$1**, **$$2**, etc., and all other variables were global. This practice is now deprecated, though still recognized, and the default visibility has changed from global to local.

The statement **print** prints a list of expressions that may include string constants such as `"hello\n"`. It does *not* print a final newline: the last expression must end with one if a newline is required.

The statement **println** works like **print**, but always supplies a following newline.

The list items printed by **print** and **println** are separated by the current value of **__OFS__** (output field separator), normally a single space.

The **printf** statement is similar to **print**, but its initial argument must be a format string conforming to a large subset of the syntax supported by Standard C's **printf**(1) statement. List item separation is controlled entirely by the format; **__OFS__** is not used. Data type length modifiers (*h*, *l*, *ll*, *L*) are not permitted, nor are *n* (dynamic field width) or *p* (pointer) format descriptors. Otherwise, the % (literal percent), *c* (character), *d* (decimal integer), *E* (uppercase exponential floating-point), *e* (lowercase exponential floating-point), *f* (fixed decimal), *G* (uppercase generalized floating-point), *g* (lowercase generalized floating-point), *i* (decimal integer), *o* (octal integer), *s* (string), *u* (unsigned integer), *x* (lowercase hexadecimal integer), and *X* (uppercase hexadecimal integer) format descriptors, with optional sign, field-width, and number-of-digits modifiers are recognized. In brief, each format descriptor is required to match this regular expression: *%(%|[-+0]?[0-9]*([.][0-9]*)?[cdeEfgGiosuxX])* .

The **sprintf** statement is similar to **printf ,** except that its result is returned as a string value, instead of being printed.

INPUT FILE SEARCH PATH

Unless input filenames specified on the command line, or in **load(***filename***)** function calls, contain a system-dependent absolute filename, **hoc** looks for them in a search path defined by the environment variable **HOCPATH**. This is an ordered list of file system directories in which to look for files. The list is colon-separated on UNIX-like systems, and semicolon-separated on systems, like Apple MacOS and Microsoft Windows, where colons are used in file paths.

For user convenience, and portability across file systems, an empty component in the directory path list stands for a default system path that includes several directories where other **hoc** are installed. Thus, **hoc** assumes a default **HOCPATH** value, if one is not already defined, of **.:**, meaning the current directory, followed by the default system path.

As a further user convenience, if an attempt to open a file fails, and the filename does not end in *.hoc*, the open is retried with that ending, allowing omission of **hoc**'s recommended file extension.

FLOATING-POINT ARITHMETIC

All arithmetic in **hoc** is done in double-precision floating point (C/C++ type **double**).

On most modern systems, this arithmetic conforms closely (or loosely) to the 1985 *IEEE 754 Standard for Binary Floating-Point Arithmetic*. This arithmetic system has numerous advantages over older designs, and has helped enormously to improve the environment for, and portability and reliability of, numerical software.

How floating-point numbers are represented

In IEEE 754 arithmetic, double-precision numbers are represented as 64-bit values, consisting of a sign bit, an 11-bit biased exponent, and a 53-bit significand. That is a total of 65 bits: the first significand bit is called a *hidden* bit, and is not actually stored. The binary point lies between the hidden bit and the stored fraction, so that for normal numbers, the significand is at least one, but less than two.

Biased, rather than explicitly signed, exponents are conventional in floating-point architectures. For IEEE 754 64-bit arithmetic, the exponent bias is 1023; that is, the true exponent is 1023 less than the stored biased value.

The smallest biased exponent (0), and the largest biased exponent (2^11 - 1 = 2047), are given special interpretation, described below for subnormals, and Infinity and NaN, respectively.

Large normal numbers

With the IEEE 754 format, the number range is approximately −1.80e+308 .. +1.80e+308, with a precision of about 15 decimal figures. The exact value of the largest floating-point number is **(1 – 2^(−53)) * 2^1024**.

Small normal numbers

The smallest *normalized* number that can be represented is about 2.23e-308, or more precisely, **2^(−1022)**, and its reciprocal is also representable, being almost exactly a quarter of the largest representable number.

Smaller subnormal numbers

The IEEE 754 Standard defines a numerically useful feature called *gradual underflow* that, when the biased exponent reaches its smallest value (0), relaxes the normalization requirement and drops the hidden bit, permitting small numbers to decrease further down to about 4.94e-324, or more precisely, **2^(−1074)**, but with loss of precision. Such numbers are called *subnormal* (formerly, *denormalized*). Not all systems support such numbers: the **hoc** function **issubnormal(x)** can be used to test whether **x** is subnormal. The reciprocal of the largest floating-point number is nonzero only if subnormal numbers are supported. Thus, you could define this **hoc** function to find out whether your system has subnormals; it returns 1 (true) if that is the case:

```
func hassubnormals() \
    return (issubnormal(1/(((1 - 2^(-53)) * 2^1023) * 2)))
```

With a predefined constant, this can also be written as

```
func hassubnormals() return (issubnormal(1/MAXNORMAL))
```

Underflow

Numbers below the smallest normalized, or when supported, the smallest subnormal, values quietly *underflow* to zero.

Machine epsilon

Another significant quantity in *any* floating-point system is known as the *machine epsilon*. This is the smallest positive number that can be added to one, and produce a sum still different from one. **hoc** provides a generalization of this, with **x** replacing **one** in the last sentence: **macheps(x)**.

In IEEE 754 arithmetic, **macheps(1)** is about 2.22e-16, or more precisely, **2^(−52)**. The negative of its base-10 logarithm is the number of decimal digits that can be represented. An error of **macheps(x)** is called an *ULP* (*U*nit in the *L*ast *P*lace). If **y** is an approximation to **x**, then with the definition

```
func errbits(x,y) \
{
    if (x == y) \
        return (0) \
    else \
        return (ceil(log2(abs((x - y)/max(x,y))/macheps(x))))
}
```

errbits(x,y) is the number of bits that are in error in **y**: that is, the base-2 logarithm of the relative error in ULPs, rounded up to the nearest integer. Incidentally, this function behaves as expected if either of its arguments are NaN (described below), or Infinity of opposite signs, even though there are no tests for those values: the result is a NaN.

One might reasonably argue for **errbits(x,y)** that the case of two Infinity arguments of like sign should also return a NaN. The current implementation does not include such a test, but doing so would require just one additional statement: `if (isinf(x) && isinf(y)) return (NAN)`.

macheps(0) is the smallest representable floating-point number, either normalized, or subnormal if supported. Thus, the test function above can be written more simply and portably (since it also works for non-IEEE 754 systems) as

```
func hassubnormals() return issubnormal(macheps(0))
```

but it will run somewhat more slowly, since the current portable implementation of **macheps(x)** involves a loop. Another simple implementation of this function uses predefined constants:

```
func hassubnormals() return (MINNORMAL > MINSUBNORMAL)
```

Special values: Infinity and NaN

IEEE 754 also defines two special values: Infinity, and NaN (not-a-number). The latter are expected to be available in two flavors: quiet and signaling, but some architectures provide only one kind. The distinction between the two NaNs is rarely significant: the Standard's intent was that quiet NaNs should be generated in numerical operations, while signaling NaNs could be used to initialize numeric variables, so that their use before assignment of a normal value could then be trapped.

Both Infinity and NaN are signed, but the sign of a NaN is usually irrelevant, and may not reflect how it was computed: some architectures only generate negative NaNs, others generate only positive ones, and a few may preserve the expected sign in the NaN produced.

hoc considers the native NaN to be positive, even if its binary encoding has a negative sign. Thus, *copysign(1.0,NaN)* returns 1.0, and *copysign(1.0,-NaN)* returns -1.0 on all systems where NaNs are available.

Signed zero

IEEE 754 has both positive and negative zero, but they compare equal. A positive zero is represented by all zero bits. A negative zero has a leading one-bit, followed by 63 zero bits.

Negative zero is generated from, e.g.,

0 / −Infinity
sqrt(−0)

In principle, you should be able to get a negative zero in any programming language by simply writing **−0**, but many compilers will convert this to positive zero. You then have to introduce a variable, assign it a zero, and negate the variable, possibly hiding the negation in an external function that simply returns its value, to foil optimizers.

In **hoc**, however, **−0** always works correctly.

Signs of numbers

In **hoc**, you can extract the sign of any value, **x**, including negative zero, Infinity, and NaN, like this:

copysign(1,x)

The result will be either +1 or −1.

Nonstop computing

Infinity and NaN are intended to provide *nonstop computing* behavior. In contrast, older architectures tended to abruptly terminate a job that computed a number too large to be stored (an *overflow*), or divided by zero. IEEE 754 arithmetic produces Infinity or NaN for these two cases, according to well-defined, and obvious, rules discussed below.

On these older systems, **hoc** tries to prevent generation of exceptional values that might otherwise terminate the job: it aborts such computations with an error message, and returns you to top level, ready for more input. On IEEE 754 systems, computation in **hoc** simply proceeds as the Standard intended.

The IEEE 754 nonstop property is exceedingly important in modern heavily-pipelined, or parallel, or superscalar, or vector, architectures, all of which have multiple operations underway at once. An interrupt to handle a floating-point exception in software is extremely costly in performance.

Properties of Infinity and NaN

Both Infinity and NaN propagate in computations, so that if they occur in intermediate results, they will usually be visible in the final results too, and alert the user to a potential problem.

Infinity behaves somewhat like a mathematical infinity:

finite / Infinity → 0
Infinity * Infinity → Infinity
Infinity^(finite or Infinity) → Infinity

NaN is produced whenever one or more operands of an arithmetic expression is a NaN, or from most numerical functions with NaN arguments, or from expressions where a limiting value cannot be determined:

Infinity − Infinity → NaN
Infinity / Infinity → NaN
0 / 0 → NaN

NaN has a unique property not shared by any other floating-point values, including Infinity: it is not equal to anything, even itself! This should be usable as a completely portable test for a NaN, even on older systems that do not have IEEE 754 arithmetic:

(x != x) is true if, and only if, **x** is a NaN.

Regrettably, compiler writers on several systems have failed to grasp this important point, and they incorrectly optimize this test to false. Thus, portable code needs to use a test function, and **hoc** provides three of them: **isnan(x)**, **isqnan(x)**, and **issnan(x)**, which return true if **x** is a NaN (of any flavor, or quiet, or signaling, respectively).

What NaNs mean for programmers

The presence of NaNs in the arithmetic system has an extremely important implication for numerical software: comparisons now have *three* outcomes, not two. The expression **(x < y)** will be true or false if neither **x** nor **y** is a NaN, but it is called *unordered* if either, or both, is a NaN. In particular, this means that it is almost always *wrong* to use a computer programming language two-branch **if − else** statement with a numerical test. Instead, there need to be additional initial tests to check for NaNs. Thus, instead of the **hoc** statement

```
if (x > y) \
    print "x is greater than y\n" \
else \
    print "x is less than or equal to y\n"
```

you should instead write

```
if (isnan(x)) \
    print "x is a NaN\n" \
else if (isnan(y)) \
    print "y is a NaN\n" \
else if (x > y) \
    print "x is greater than y\n" \
else \
    print "x is less than or equal to y\n"
```

Since **if − else** statements are very common in software, but most programmers, and computer textbook authors, are not sufficiently familiar with IEEE 754 arithmetic, you should expect that most existing software, and textbook examples, will fail to behave consistently, or correctly, when dealing with NaN, and possibly also Infinity.

There have been some major disasters, such as the failure of the Ariane satellite launch in West Africa, the failure of Patriot missiles in the Gulf War, and a U.S. nuclear aircraft carrier sitting dead in the water for six hours, all attributed to computer programmers who lacked sufficiently understanding of computer arithmetic. Arithmetic really does matter!

Numerical software often contains convergence tests of the form

while (tolerance is not reached)
  reduce the tolerance

If a NaN ever appears in the **while** expression, the test will never be satisfied, and the program will be in an infinite loop. Even famous libraries like EISPACK and LINPACK have routines that will never return because of loops caused by NaNs. [In fairness, both of those libraries were developed before IEEE 754 arithmetic existed, but CDC and Cray machines of that era had special values similar to Infinity and NaN, so even then, there were systems where the code could endlessly loop.]

Vendor-provided floating-point systems and run-time libraries are not always entirely reliable in their handling of signed zero, Infinity, and NaN, and portable programs like **hoc** can help to ferret out implementation differences, and errors that should be reported to the vendors. As noted earlier, signed zero is often botched by compiler writers, and two functions commonly available in most programming languages, **max(x,y)** and **min(x,y)**, in particular are badly done. Their simple implementations use a two-branch conditional like this one for **max(x,y)**: `if (x > y) return x else return y`. If either argument is a NaN, then the test will fail, and the second argument will be returned, leading to inconsistent nonsense like **max(1,NaN) → NaN** but **max(NaN,1) → 1**. The C and C++ languages lack such functions (users are expected to write them as macros), but Fortran and many other languages have them. In the fall of 2001, tests of 61 Fortran compilers on 15 different UNIX platforms showed that *all* fail to behave consistently for **max(x,y)** and **min(x,y)**.

Unsupported IEEE 754 features

Finally, there are two additional features of IEEE 754 arithmetic that are not yet supported by this version of **hoc**, but will be in future releases:

(1) access to floating-point status flags, so that you can tell after the fact whether a computation encountered any exceptional conditions, and

(2) access to rounding control, which determines whether rounding is to minus Infinity, zero, nearest, or plus Infinity. The default is always round-to-nearest.

Once rounding control is available, **hoc** could, in principle, be extended to support interval arithmetic, in which each numeric operation produces upper and lower bounds for the result. Of course, a proper implementation would also require such support in all of the mathematical functions in the C/C++ run-time library, and such support is lacking almost everywhere.

HELP SYSTEM

One of the files that **hoc** normally loads on startup contains an extensive help system. Each named constant, variable, function and procedure has an associated function, **help_NAME( )**, where **NAME** is the object name. Help is also available on each of the **hoc** language statements, and on related topics. For an introduction, run **help( )**, and for a detailed list of what help functions are available, invoke **help_help( )**. To display the entire help system, invoke **help_all( )**.

Users are encouraged to follow these help convention with their own **hoc** code.

The entire help corpus is intentionally *external* to **hoc** itself, to facilitate modification, partial replacement, and internationalization, as discussed in the next section.

INTERNATIONALIZATION

The **hoc** help system can be readily extended to support documentation in languages other than English, and early releases contain limited prototype text in several languages.

Changing the language alters only documentation and program messages: the basic **hoc** language remains unchanged, and English-centric, just as do virtually all computer programming languages.

Selecting a language

An alternate language is selected at run-time by defining any one of three environment variables: **LC_ALL**, **LC_MESSAGES**, or **LANG**, just as described for other programming languages in **locale**(1). These variables take values of a locale code, the values of which you can list by

```
locale -a | sort -f
```

You could thus launch a German version of **hoc** like this:

```
env LANG=de hoc
```

Environment variables, rather than command-line options, control the locale selection, because it is likely that most individuals will want to choose a fixed locale, and that can be done once and for all in user login files, and also because several UNIX library functions access the locale environment variables to guide their behavior. UNIX users could also create convenient shell aliases, e.g., in **csh**(1)/ **tcsh**(1) syntax,

```
alias hoc-da 'env LANG=da hoc \!*'
alias hoc-de 'env LANG=de hoc \!*'
alias hoc-fr 'env LANG=fr hoc \!*'
...
```

What if you have no locale support?

Virtually all UNIX vendors today provide locale support, but they usually require installation of one or more additional software packages that your system manager may have omitted, but is probably willing to install on request.

Locale support is usually present in one of these directories; besides using the **locale**(1) command as shown in the previous subsection, you can run **ls**(1) on the appropriate one of them to see what locales are installed on your system:

/usr/share/locale	Apple Darwin (MacOS X), FreeBSD, GNU/Linux (all architectures)
/usr/lib/nls/loc	Compaq/DEC Alpha, IBM AIX
/usr/share/i18n/locales	GNU/Linux (all architectures)
/usr/lib/nls/loc/locales	Hewlett-Packard HP-UX

/usr/lib/locale SGI IRIX, Sun Solaris

What the locale affects

Normally, changing the locale affects more than just text: dates, monetary formats, numbers, and sort order all change. However, for now, in the interests of simplicity, and cross-platform and cross-locale consistency, **hoc** sets the locale categories for **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** to their traditional (English/American) values. Changes will be needed in future versions of **hoc** to support other values of these categories; some of that support is already available, as shown in the next subsection.

Changing the locale inside hoc programs

Locale categories can be set in the environment from *inside* **hoc** programs to control calendar date and time formatting by the **strftime( )** function:

```
# Show time in the default locale:
hoc> strftime("%c",systime())
Fri Dec 21 15:18:14 2001

# Switch to Portuguese: ISO8859-1 (Latin-1) encoding:
hoc> old_lc_time = putenv("LC_TIME", "pt")
hoc> strftime("%c",systime())
sex 21 dez 2001 03:17:29 PM MST

# Restore the original locale:
hoc> ignore = putenv("LC_TIME", old_lc_time)
```

The current locale setting can be saved and restored as shown. Less desirably, the value `"C"` resets it to the C/C++ default of English.

The locale code is interpreted as the name of a subdirectory in which to find a localized version of any system file that **hoc** loads at startup time. For example, in a Danish locale, it will load the English file, `help.hoc`, and then the Danish file, `da/help.hoc`, from the **hoc** system installation directory, provided that the localized file exists. Otherwise, **hoc** is silent about its absence.

Changing the language of internal messages

The **hoc** program contains a number of messages that are hard-coded in English. Any, or all, of these can be replaced at run time by assignments to special variables named with the reserved seven-character prefix **___msg_** (yes, there are *three* leading underscores) used to identify translation variables.

These variables are normally only set in the *translations.hoc* files in the **hoc** system directory tree, but they can also be set by user programs as well, unless they have been defined as permanent constants.

See the comments in those files for further documentation. Except for translation work, it should never be necessary for ordinary users to reference or modify these variables.

Character set constraints

The significant constraint is that characters must be representable in 8-bit character sets, such as the dozen or so ISO8859-*n* sets that supply characters needed for European languages, or the Unicode (also known as ISO10646-1) UTF-8 variable-byte-count encoding of potentially two million or so symbols used in the world's writing systems. In addition, the **hoc** user must be running the program in an environment capable of such display.

Changing screen display fonts

In a UNIX system, you might first scan the voluminous output of **xlsfonts**(1) to find out what fonts are available for your window system, and then launch a terminal window like this:

```
xterm -fn \
    -adobe-courier-medium-r-normal--14-100-100-100-m-90-iso8859-1 &
```

to get a 14pt font with all of the characters needed for ISO8859-1 (Latin 1, handling most of the languages of Western Europe, and many others, such as Hawaiian, Indonesian, and Swahili).

Your system manager may be able to tell you about additional window system fonts that may also be available, but are not loaded by default. For example, at the maintainer's site, there is a large collection of Asian and European fonts installed in the **emacs**(1) editor tree. To add, say, the European collection, in a shell window type

```
xset fp+ /usr/local/share/emacs/fonts/European
xset fp rehash
```

The new fonts will then be available, and will be listable by **xlsfonts**(1). You can make those additions permanent by adding those two commands to your *$HOME/.xinitrc* or *$HOME/.xsession* file; the name is platform-dependent, so the best choice is to make them identical, with one a symbolic link to the other.

Use

```
xset q
```

to find out what font directories are currently in the font search path.

Each X Window System font directory has a `fonts.dir` text file that maps short file names to long font names. There is sometimes also a `fonts.alias` text file to provide short aliases for the otherwise rather long font names used in the X Window System. You can scan those files to see what is available.

Recent versions of **xterm**(1) have a special option, **–u8**, to handle UTF-8 multibyte encoding, but you then need to use a font with the corresponding character repertoire:

```
xterm -u8 -fn \
    -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso10646-1 &
```

Documentation for hoc in other languages

Internationalized documentation will usually augment, rather than replace, the English documentation. That way, translations can be developed incrementally. Thus, in a French environment, **help( )** responds in English, while output from **aide( )** is in French. On startup, **hoc** will then usually display a greeting in two languages: English, and the local one. Here is what this looks like in the French locale:

```
% env LANG=fr hoc
---------------------------------------------------------------
Welcome to the extensible high-order calculator, hoc.
This is hoc version 7.0.0.beta [15-Dec-2001].
Type help() for help, news() for news, and author() for author
information.
This system supports IEEE 754 floating-point arithmetic.
---------------------------------------------------------------
---------------------------------------------------------------
Bienvenue à la calculatrice, hoc.
C'est la version 7.0 du 15 décembre 2001.
Taper aide() pour de l'assistance, nouvelles() pour des
nouvelles, et auteur() pour des renseignements sur les
auteurs.
Cet ordinateur supporte l'arithmétique en virgule flottante du
standard IEEE 754.
---------------------------------------------------------------
```

The maintainer will be grateful for contributions of additional translations of **hoc** help files and internal messages!

HOC SUPPORT IN GNU EMACS

When **hoc** is installed properly, it adds a new library, *hoc.el*, to the *emacs/site-lisp* directory, which should always be included in the **emacs**(1) *load-path* variable (in an editor session, type `C-h vload-path` to display it).

By suitable manual edits to the *site-init.el* file in that directory, your system manager could make **hoc-mode** support automatically available, but the **hoc** installation process cannot safely do that automatically.

You can test whether this has been done at your site by visiting a new file with extension *.hoc*; if the **emacs**(1) mode line shows `(hoc ...)`, instead of something else, like `(fundamental ...)`, then you need do nothing more: **hoc-mode** is already fully installed.

Otherwise, in order to avoid the need for tedious manual loading of the **hoc** support in **emacs**(1), add this snippet of Emacs Lisp code at the end of your *$HOME/.emacs* initialization file:

```
(if (string-lessp (substring emacs-version 0 2) "19") ; earlier than 19.x
    (progn
      (setq auto-mode-alist
            (cons (cons "\.hoc$" 'hoc-mode) auto-mode-alist))
      (autoload 'hoc-mode "hoc"
        "Enter hoc mode." t nil))
  (progn
    (if (not (assoc "\.hoc$" auto-mode-alist))
        (setq auto-mode-alist
              (cons (cons "\.hoc$" 'hoc-mode) auto-mode-alist)))
    (autoload 'hoc-mode "hoc"
      "Enter hoc (high-order calculator) mode." t nil)))
```

There are two sections in this code, one for (now very old) **emacs**(1) versions before 19.x, and the other for all newer versions. They add a binding between files with extension *.hoc* and **hoc-mode** in **emacs**(1), and arrange for the *hoc.el* library to be loaded the first time that it is required.

Two additional functions are provided to ease the task of creating help procedures: **hoc-printify** and **hoc-unprintify**. Both operate on the region, converting text to **print** statements, or the reverse.

DESCRIPTIONS OF BUILT-IN FUNCTIONS AND PROCEDURES

These descriptions are taken from the output of the corresponding **help_xxx( )** functions, and, apart from font differences, are intended to be identical to them. The **help_xxx( )** functions are considered to be the definitive documentation of each function.

In the following descriptions, square brackets on number ranges indicate that the endpoint is *included*; parentheses indicate that the endpoint is *excluded*.

abort(message) — **abort(message)** prints **message**, then aborts evaluation of the current expression, returning to top-level without further processing of the remainder of the current statement or function/procedure call chain. The message should include the name of the function calling **abort( )**, since there is currently no function call traceback, and end with a newline.

abs(x) — **abs(x)** returns the absolute value of **x**.

acos(x) — **acos(x)** returns the arc cosine of **x**. **x** must be in [−1...+1].

acos(x) — **acos(x)** returns the arc cosine of **x**. **x** must be in [−1...+1].

acosh(x) — **acosh(x)** returns the inverse hyperbolic cosine of **x**. **x** must be outside the interval (−1...+1).

asinh(x) — **asinh(x)** returns the inverse hyperbolic sine of **x**.

atan(x) — **atan(x)** returns the arc tangent of **x**.

atanh(x) — **atanh(x)** returns the inverse hyperbolic tangent of **x**.

author( ) — **author( )** prints information about the program authors.

cbrt(x) — **cbrt(x)** returns the cube root of **x**.

cd(*s*) — **cd(*s*)** changes the current working directory to that named by the string *s*, updates the environment variable **PWD** to that name, and returns that name.

See also **help_dirs( )**, **help_popd( )**, **help_pushd( )**, **help_pwd( )**, and **help_xd( )**.

ceil(x) — **ceil(x)** returns the smallest integer greater than or equal to **x**.

char(n) — **char(n)** returns a one-character string containing the character whose ordinal value in the host character set is **n**. Characters are always considered *unsigned*. Thus, in an ASCII or ISO 8859-n or Unicode character set, **char(65)** returns **"A"**, and both **char(255)** and **char(-1)** return **"\xff"**.

hoc strings are internally terminated by a NUL character, so **char(0)** is equivalent to an empty string, **""**, and **("X" char(0) "Y")** evaluates to **("X" "" "Y")** which in turn reduces to **"XY"**.

class(x) **class(x)** returns a numeric value (available as a predefined constant) indicating into which of these ten classes **x** falls:

CLASS_NEGINF	negative infinity
CLASS_NEGNORMAL	negative normal
CLASS_NEGSUBNORMAL	negative subnormal
CLASS_NEGZERO	negative zero
CLASS_POSINF	positive infinity
CLASS_POSNORMAL	positive normal
CLASS_POSSUBNORMAL	positive subnormal
CLASS_POSZERO	positive zero
CLASS_QNAN	quiet NaN
CLASS_SNAN	signaling NaN

An eleventh value is reserved to flag classification failure:

CLASS_UNKNOWN	should never happen

copysign(x,y) **copysign(x,y)** returns a value with the magnitude of **x**, and the sign of **y**.

cos(x) **cos(x)** returns the cosine of **x** (**x** in radians). Expect severe accuracy loss for large **|x|**.

cosd(x) **cosd(x)** returns the cosine of **x** (**x** in degrees). Expect severe accuracy loss for large **|x|**.

cosh(x) **cosh(x)** returns the hyperbolic cosine of **x**.

cpulimit(t) **cpulimit(t)** sets the CPU time limit from now to an additional **t** seconds, sets the system variable **__CPU_LIMIT__** to **t**, and returns the current CPU time limit, which is always measured from the *start* of the job.

If the limit is exceeded, execution of the current expression is aborted, control returns to the top-level interpreter, and the time limit is incremented by the current value of **__CPU_LIMIT__**.

Although **t** may be fractional, on most operating systems, the time limit is an integer, so **t** will be rounded up internally to the nearest integer before setting the time limit.

If resource usage and limits are not supported on the current platform, this function has no effect, other than setting **__CPU_LIMIT__**, and returning Infinity.

By default, there is no time limit for the job (although some operating systems may impose such limits).

Negative, zero, and NaN arguments are treated like Infinity.

NB: This function is *experimental*, and may be withdrawn in future versions.

defined(symbol) **defined(symbol)** returns 1 if **symbol** is defined, and 0 if not.

Programming note: This function can be used in **hoc** libraries to provide default values of variables, for example,

```
if (!defined(seed)) seed = 123456789
```

delete(symbol) **delete(symbol)** returns 1 if **symbol** was successfully deleted, and 0 if not. When a symbol is deleted, its value is no longer available, as if it had never been defined.

Most user-defined symbols can be deleted, but **hoc** kernel symbols, and user-defined immutable symbols, cannot.

dirs( ) **dirs( )** prints the current directory stack, with the most recent directory first.

See also **help_cd( )**, **help_popd( )**, **help_pushd( )**, **help_pwd( )**, and **help_xd( )**.

double(x) **double(x)** returns the value of **x** converted to double precision, and then back to working precision.

erf(x) **erf(x)** returns the error function of **x**.

erfc(x) **erfc(x)** returns the complementary error function of **x**.

errbits(x,y) **errbits(x,y)**, with **y** an approximation to **x**, returns the number of bits that **y** is in error by.

eval(string) **eval(string)** pushes its argument string, which must contain valid **hoc** code, onto the input stack so that it will be evaluated next. The size of the input stack is limited only by available memory.

This function makes it possible for **hoc** programs to construct new **hoc** code on-the-fly and then run it.

exp(x) **exp(x)** returns the exponential function of **x**, **E^x**.

expm1(x) **expm1(x)** returns the exponential function of **x**, less 1: **E^x – 1**.

For small **x**, **exp(x)** is approximately 1, so there is serious subtraction loss in directly using **exp(x) – 1**; **expm1(x)** avoids this loss.

From Sun Solaris documentation: ‘‘*The* **expm1( )** *and* **log1p( )** *functions are useful for financial calculations of* **((1 + x)^n – 1) / x**, *namely:*

expm1(n * log1p(x))/x

when **x** *is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.*’’

exponent(x) **exponent(x)** returns the base-2 exponent of **x**, such that

x == significand(x) * 2^exponent(x)

where **|significand(x)|** is in [1...2).

For IEEE 754 arithmetic, normal numbers have **exponent(x)** in [-1022...1023] and subnormal numbers, if supported, have **exponent(x)** in [-1074...1023].

WARNING: The power **2^exponent(x)** will underflow to zero for IEEE 754 subnormal numbers, so for such numbers, the right-hand side must be computed with suitable scaling, like this:

(significand(x) * 2^(exponent(x) + 52)) * 2^(-52)

factorial(n) **factorial(n)** returns **n! = n*(n–1)*(n–2)*. . .*1**, where **1! == 0! == 1**, by definition. Negative arguments generate a call to **abort( )**.

floor(x) **floor(x)** returns the greatest integer less than or equal to **x**.

fmod(x,y) **fmod(x,y)** returns the remainder of the division of **x** by **y**.

ftoh(x) **ftoh(x)** returns a hexadecimal string containing the native floating-point representation of **x**.

For readability, a separating underscore is inserted between groups of eight hexadecimal digits.

This function is the inverse of **htof(s)**.

gamma(x) **gamma(x)** returns the Gamma (generalized factorial) function of **x**.

gcd(x,y) **gcd(x,y)** returns the greatest common divisor of **x** and **y**.

getenv(envvar) **getenv(envvar)** returns the string value of the environment variable **envvar**, or an empty string if it is not defined.

hexfp(x) **hexfp(x)** returns a string containing the hexadecimal floating-point representation of **x**, in the form

"+0x1.hhhhh...p+ddddd"

Trailing zeros in the fraction, and leading zeros in the exponent, are dropped, and the sign is always included.

See also **help_hexint( )**, **help_number( )**, and **help_string( )**.

hexint(x) **hexint(x)** returns a string containing the hexadecimal integer representation of **x**, if that is possible, in the form

"+0xhhhhh..."

Leading zeros are dropped, and the sign is always included.

If **x** is too big to represent as an exact integer, then the floating-point representation, **hexfp(x)**, is returned instead.

See also **help_hexfp( )**, **help_number( )**, and **help_string( )**.

htof(s) **htof(s)** returns a floating-point number corresponding to the native representation in the hexadecimal string **s**.

Nonhexadecimal digits in **s** are ignored.

This function is the inverse of **ftoh(x)**.

hypot(x,y) **hypot(x,y)** function computes the length of the hypotenuse of a right-angled triangle, **sqrt(x^2 + y^2)**, but without accuracy loss or range limitation from premature overflow or underflow.

This function has possibly unexpected behavior for exceptional arguments: when either argument is Infinity, then the result is Infinity, *even if* the other argument is a NaN! The explanation is found on the 4.3BSD manual page:

> . . . programmers on machines other than a VAX (it has no infinity) might be surprised at first to discover that **hypot(+infinity,NaN) = +infinity**. This is intentional; it happens because **hypot(infinity,v) = +infinity** for all **v**, finite or infinite. Hence **hypot(infinity,v)** is independent of **v**. Unlike the reserved operand on a VAX, the IEEE NaN is designed to disappear when it turns out to be irrelevant, as it does in **hypot(infinity,NaN)**. . . .

ichar(s) **ichar(s)** returns the ordinal value of the first character in the string **s**. Characters are always considered *unsigned*. Thus, in an ASCII or ISO 8859-n or Unicode character set, **ichar("ABC")** returns 65, and **ichar("\xff")** returns 255. **ichar("")** returns 0, because **hoc** strings are internally terminated by a NUL (zero-valued) character.

ilogb(x) **ilogb(x)** returns the exponent part of **x**, that is, **int(log2(x))**.

index(s,t) **index(s,t)** returns the index of string **t** in string **s**, counting from 1, or 0 if **t** is not found in **s**.

int(x) **int(x)** returns the integer part (truncated toward zero) of **x**.

isfinite(x) **isfinite(x)** returns 1 (true) if **x** is finite and otherwise, 0 (false).

isinf(x) **isinf(x)** returns 1 (true) if **x** is Infinite, and otherwise, 0 (false).

isnan(x) — **isnan(x)** returns 1 (true) if **x** is a NaN, and otherwise, 0 (false).

isnormal(x) — **isnormal(x)** returns 1 (true) if **x** is finite and normalized and not subnormal, and otherwise, 0 (false).

isqnan(x) — **isqnan(x)** returns 1 (true) if **x** is a quiet NaN, and otherwise, 0 (false).

On some architectures (e.g., Intel x86 and MIPS), there is only one type of NaN. **isqnan(x)** is then defined to return **isnan(x)**.

issnan(x) — **issnan(x)** returns 1 (true) if **x** is a signaling NaN, and otherwise, 0 (false).

On some architectures (e.g., Intel x86 and MIPS), there is only one type of NaN. **issnan(x)** is then defined to return **isnan(x)**.

You can test whether your system has both quiet and signaling NaNs like this: **issnan(NaN)**. The result is 0 (false) if distinct NaN types are available, and 1 (true) if not.

issubnormal(x) — **issubnormal(x)** returns 1 (true) if **x** is subnormal (formerly, denormalized), and otherwise, 0 (false).

J0(x) — **J0(x)** returns the Bessel function of the first kind of order 0 of **x**.

J1(x) — **J1(x)** returns the Bessel function of the first kind of order 1 of **x**.

Jn(n,x) — **Jn(n,x)** returns the Bessel function of the first kind of integral order **n** of **x**.

lcm(x,y) — **lcm(x,y)** returns the least common multiple of **int(x)** and **int(y)**.

ldexp(x,y) — **ldexp(x,y)** returns **x * 2^(int(y))**.

lgamma(x) — **lgamma(x)** returns the natural logarithm of **gamma(x)**.

Because **gamma(x)** has poles at zero and at negative integer values, and grows factorially with increasing **x**, it reaches the floating-point overflow limit fairly quickly. For 64-bit IEEE 754 arithmetic, this happens at approximately **x = 206.779**. However, **lgamma(x)** is representable almost to the overflow limit. In 64-bit IEEE 754 arithmetic, this happens at approximately **x = 2.55e+306** (the overflow limit is 1.80e+308).

Unfortunately, there is mathematically-unavoidable accuracy loss when **gamma(x)** is computed from **exp(lgamma(x))**, so you should avoid the logarithmic form unless you really need large arguments that would cause overflow.

ln(x) — **ln(x)** returns the natural (base-E) logarithm of **x**.

load(*"filename"*) — **load(*"filename"*)** reads input from the specified file. The file can be prepared by hand, or by the **save( )** command.

See the **INPUT FILE SEARCH PATH** section below for details on how **hoc** finds input files.

Loaded files can themselves contain **load( )** commands, with nesting up to some unknown limit imposed by the host operating system on the maximum number of simultaneously-open files for a process, user, or the entire system.

This command can be disabled for security reasons by the command-line **−noload** or **−secure** options.

The return value is an empty string on success, and otherwise, an error message.

log(x) — **log(x)** returns the natural (base-E) logarithm of **x**.

log10(x) — **log10(x)** returns the logarithm to the base 10 of **x**.

log1p(x) — **log1p(x)** returns **log(1 + x)**, but without accuracy loss for small **|x|**. **x** must be in (−1. . .infinity].

log2(x) **log2(x)** returns the logarithm to the base 2 of **x**.

logfile(filename) **logfile(filename)** logs the session on the specified file, which, for security reasons, *must* be a new file. It is a normal text which you can edit, print, and view.

Input is recorded verbatim. Output is recorded in comments. This permits the logfile to be read by **hoc** later, allowing a session to be replayed.

If a logfile is already opened, it is closed before opening the new one.

Logging may be turned on and off with **logon( )** and **logoff( )**, and can be entirely disabled for security reasons by the command-line **−no-logfile** option.

The return value is an empty string on success, and otherwise, an error message.

logoff( ) **logoff( )** suspends logging to any open log file. It is *not* an error if there is no current log file.

logon( ) **logon( )** restores logging to any open log file. It is *not* an error if there is no current log file.

macheps(x) **macheps(x)** returns the generalized machine epsilon of **x**, the smallest number which, when added to **x**, produces a sum that still differs from **x**: **(x + macheps(x)) != x**.

macheps(1) is the normal machine epsilon.

macheps(−x) is **macheps(x)/base**, or equivalently, the smallest number that can be subtracted from **x** with the result still different from **x**.

macheps(0) is the smallest representable floating-point number. Depending on the host system, it may be a normal number, or a subnormal number (invoke **help_subnormal( )** for details).

max(x,y) **max(x,y)** returns the larger of **x** and **y**.

If *either* argument is a NaN, the result is a NaN.

maxnormal( ) **maxnormal( )** returns the maximum positive normal number.

min(x,y) **min(x,y)** returns the smaller of **x** and **y.**

If *either* argument is a NaN, the result is a NaN.

minnormal( ) **minnormal( )** returns the minimum positive normal number.

minsubnormal( ) **minsubnormal( )** returns the minimum positive subnormal number. If subnormals are not supported, then it returns the minimum normal number instead.

msg_translate(msg) **msg_translate(msg)** looks up the message string, **msg**, in **hoc**'s translation tables, and if a nonempty translation exists, returns that translation; otherwise, it returns its argument, **msg**.

Please use this function in your own **hoc** code to ensure that your messages can be translated to other languages without any changes whatsoever to your code.

nearest(x,y) **nearest(x,y)** returns the next different machine number nearest **x**, in the direction of the infinity with the same sign as **y.**

nextafter(x,y) **nextafter(x,y)** returns the nearest machine number nearest **x**, in the direction of the infinity with the same sign as **y**.

nint(x) **nint(x)** returns the nearest integer to **x**, rounding away from zero in case of a tie.

now( ) **now( )** returns the current date and time, in the standard UNIX form

```
"Wed Jul  4 14:57:51 2001"
```

If the month day has only one digit, then it is preceded by an extra space.

number(s) **number(s)** converts the string **s** to a number and returns it.

s should contain either a hexadecimal floating-point number, a hexadecimal integer, a decimal floating-point number, a decimal integer, or a representation of NaN or Infinity.

If **s** contains a number followed by unrecognizable text, the number is converted and returned, and the following text is silently ignored. Otherwise, the return value is 0, and the text is silently ignored. Thus, **number("123abc")** returns 123, and **number("abc")** returns 0.

This function is an inverse of **hexfp( )**, **hexint( )**, and **string( )**:

> **number(hexfp(x)) == x** [for all numeric **x**]
> **number(hexint(x)) == x** [for all numeric **x**]
> **number(string(x)) == x** [for all numeric **x**]

See also **help_hexint( )**, **help_hexfp( )**, and **help_string( )**.

popd( ) **popd( )** removes the top-most element from the current directory stack, makes it the current directory, and calls **dirs( )** to print the updated stack.

See also **help_cd( )**, **help_dirs( )**, **help_pushd( )**, **help_pwd( )**, and **help_xd( )**.

printenv(pattern) **printenv(pattern)** prints the names and values of all environment variables whose names match **pattern**, sorted in strict lexicographic order.

Matching is similar to UNIX shell pattern matching: asterisk (*) matches zero or more characters, and query (?) matches any single character. A square-bracketed list of characters, and/or hyphen-separated character ranges, matches any character in that list. A right bracket can be in the list only if it appears first. Thus, [A-Za-z0-9] matches an English letter or digit, and [][] matches a square bracket.

To match all environment variables, use **printenv("*")**.

protect(s) **protect(s)** returns a copy of the string **s** with all nonprintable characters represented as escape sequences.

pushd(s) **pushd(s)** tries to make the directory named by the string **s** the new current working directory, and if that was successful, makes that directory the new top of the current directory stack, and calls **dirs( )** to print the updated stack.

See also **help_cd( )**, **help_dirs( )**, **help_popd( )**, **help_pwd( )**, and **help_xd( )**.

putenv(envvar,newval)

putenv(envvar,newval) replaces the current string value of the environment variable **envvar** with **newval**, and returns its old value.

This affects subsequent calls to **getenv( )**, but does *not* affect the environment of the parent process.

You can use this function to set locale environment variables that control the output of dates and times, in order to get internationalized output from **strftime( )**.

pwd( ) returns the name of the current working directory. That name is also available in the environment as **getenv("PWD")**.

See also **help_cd( )**, **help_dirs( )**, **help_popd( )**, **help_pushd( )**, and **help_xd( )**.

rand( ) **rand( )** returns a pseudo-random number uniformly distributed on (0. . .1). Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

See **help_randint( )** for uniformly-distributed integers in an interval, and **help_randl( )** for logarithmically-distributed pseudo-random numbers.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

randint(x,y)
: **randint(x,y)** returns a pseudo-random integer uniformly distributed on **[int(x)...int(y)]**. Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

randl(x)
: **randl(x)** returns a pseudo-random number logarithmically distributed on **(1,exp(x))**. Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

This function can be used to generate logarithmic distributions on any interval: **a*randl(ln(b/a))** is logarithmically distributed on **(a...b)**.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

remainder(x,y)
: **remainder(x,y)** returns the remainder **r = x − n*y**, where **n** is the integral value nearest the exact value **x/y**. When **|n − x/y| = 1/2**, the value of **n** is chosen to be even.

rint(x)
: **rint(x)** returns the integral value nearest **x** in the direction of the current IEEE 754 rounding mode.

rsqrt(x)
: **rsqrt(x)** returns the reciprocal square root, **1/sqrt(x)**.

save(filename,pattern)
: **save(filename,pattern)** saves the state of the current session in the specified file, which, for security reasons, *must* be a new file.

Only symbols whose names match **pattern** are saved.

Matching is similar to UNIX shell pattern matching: asterisk (*) matches zero or more characters, and query (?) matches any single character. A square-bracketed list of characters, and/or hyphen-separated character ranges, matches any character in that list. A right bracket can be in the list only if it appears first. Thus, [A-Za-z0-9] matches an English letter or digit, and [][] matches a square bracket.

To match all symbols, use **save(filename,"*")**.

Symbols are output in strict lexicographic order.

Reserved symbol names (those beginning with two or more underscores) are not saved. Predefined immutable names are also excluded.

The saved file is a normal text file that can be later read by **hoc** on any platform.

[NB: A temporary implementation restriction also excludes user-defined immutable names, and all functions and procedures.]

This command can be disabled for security reasons by the command-line **−no-save** option.

The return value is an empty string on success, and otherwise, an error message.

scalb(x,y)
: **scalb(x,y)** returns **x * 2^(int(y))**.

second( )
: **second( )** returns the CPU time in job seconds since some fixed time in the past. Take the difference of two bracketing calls to get the elapsed CPU time for a block of code. For example,

```
PREC = 3
x = 1
t = second()
```

```
for (k = 1; k < 1000000; ++k) x *= 1
second() - t
4.73
```

set_locale(localecode) **set_locale(localecode)** loads the locale files for the locale identified by **localecode**. This must correspond to a subdirectory of the **hoc** system directory, which is

```
/usr/local/share/lib/hoc/hoc-7.0.6.beta
```

in this installation.

Since **set_locale( )** is a long name, up to three shorthand procedures are provided for each language: the two-letter country code, the native name for the language, and the English name for the language. Thus, **da( )**, **dansk( )**, and **danish( )** all switch to the Danish locale, and **en( )**, **engelsk( )**, and **english( )** switch to the default English locale.

setrand(x) **setrand(x)**, where **x** should be a large integer, sets the seed of the pseudo-random number generator to **x**, and returns the old seed.

As a special case, when **x** is zero, **x** is ignored, and a new seed is constructed from a random number multiplied by either the calendar time (if available), or the process number (if available), or the next pseudo-random number.

If **setrand(x)** is never called, then **rand( )**, **randint( )**, and **randl(x)** will each return the same sequence of pseudo-random numbers: see **help_rand( )**, **help_randint( )**, and **help_randl( )**.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

significand(x) **significand(x)** returns the significand of **x**, **s,** such that **x = s * 2^n**, with **s** in [1,2), and **n** an integer.

See **help_exponent( )** for how to extract the exponent, **n**.

sin(x) **sin(x)** returns the sin of **x** (**x** in radians). Expect severe accuracy loss for large **|x|**.

sind(x) **sind(x)** returns the sin of **x** (**x** in degrees). Expect severe accuracy loss for large **|x|**.

single(x) **single(x)** returns the value of **x** converted to single precision, and then back to working precision.

sinh(x) **sinh(x)** returns the hyperbolic sin of **x**.

snan() returns a distinct signaling NaN on each call, up to some architectural limit, after which, the values cycle. At least 2^23 different values can be returned. The only way to distinguish between them is to examine their bit representations with **ftoh**().

By contrast, the predefined constants SNAN and SNaN have fixed bit patterns.

See also **ftoh**(), **htof**(), **NaN**, **QNaN**, and **SNaN**.

sqrt(x) **sqrt(x)** returns the square root of **x**. **x** must be in [−0. . .Infinity].

Special case: **sqrt(−0)** → **−0**.

strftime(format,time) **strftime(format,time)** converts a numeric time measured in seconds since the epoch (usually obtained from **systime( )**) to a formatted string determined by one or more of these format items:

%A the locale's full weekday name.

%a the locale's abbreviated weekday name.

%B the locale's full month name.

%b the locale's abbreviated month name.

%c the locale's appropriate date and time representation.

%d the day of the month as a decimal number (01−31).

%H the hour (24-hour clock) as a decimal number (00–23).

%I the hour (12-hour clock) as a decimal number (01−12).

%j the day of the year as a decimal number (001−366).

%M the minute as a decimal number (00–59).

%m the month as a decimal number (01−12).

%p the locale's equivalent of either ''AM'' or ''PM''.

%S the second as a decimal number (00–60).

%U the week number of the year (Sunday as the first day of the week) as a decimal number (00–53).

%W the week number of the year (Monday as the first day of the week) as a decimal number (00–53).

%w the weekday (Sunday as the first day of the week) as a decimal number (0–6).

%X the locale's appropriate time representation.

%x the locale's appropriate date representation.

%Y the year with century as a decimal number.

%y the year without century as a decimal number (00–99).

%Z the time zone name.

%% is replaced by '%'.

string(x) **string(x)** returns a string containing the decimal representation of **x**, either in integer form (if **x** is exactly representable that way), or in floating-point form.

See also **help_hexfp( )**, **help_hexint( )**, and **help_number( )**.

substr(s,start,len) **substr(s,start,len)** returns a substring of string **s** beginning at character **start** (counting from 1), of length at most **len**. If **start** is outside the string, it is moved to the nearest endpoint, *without* adjusting **len**. Fewer than **len** characters will be returned if the substring extends outside the original string.

symnum(s) **symnum(s)** converts the string **s** to a symbol naming a numeric variable, which must exist. It may then be used almost like any numeric variable name, wherever its value is taken, but it cannot be used to define a symbol, such as on the left-hand side of an assignment statement.

symstr(s) **symstr(s)** converts the string **s** to a symbol naming a string variable, which must exist. It may then be used almost like any string variable name, wherever its value is taken, but it cannot be used to define a symbol, such as on the left-hand side of an assignment statement.

systime( ) **systime( )** returns the calendar time in seconds since the epoch. On UNIX systems, the epoch starts on January 1, 1970 00:00:00 UTC. Other operating systems make different choices. It can be converted to a formatted time string with **strftime( )**.

tan(x) **tan(x)** returns the tangent of **x** (**x** in radians). Expect severe accuracy loss for large **|x|**.

tand(x) — **tand(x)** returns the tangent of **x** (**x** in degrees). Expect severe accuracy loss for large **|x|**.

tanh(x) — **tanh(x)** returns the hyperbolic tangent of **x**.

tolower(s) — **tolower(s)** returns a copy of string **s** with uppercase letters converted to lowercase, and all other characters unchanged.

Which characters are considered uppercase depends on the locale. On UNIX, this is determined by the **LC_CTYPE** environment variable.

toupper(s) — **toupper(s)** returns a copy of string **s** with lowercase letters converted to uppercase, and all other characters unchanged.

Which characters are considered lowercase depends on the locale. On UNIX, this is determined by the **LC_CTYPE** environment variable.

trunc(x) — **trunc(x)** returns the integer part of **x**, with the fractional part discarded.

unordered(x,y) — **unordered(x,y)** returns 1 (true) if **x** or **y** is unordered with respect to the other (that is, at least one of them is a NaN), and otherwise, 0 (false).

who(pattern) — **who(pattern)** prints all symbols whose names match the **pattern** string, grouped by category, and sorted lexicographically within each category.

Matching is similar to UNIX shell pattern matching: asterisk (*) matches zero or more characters, and query (?) matches any single character. A square-bracketed list of characters, and/or hyphen-separated character ranges, matches any character in that list. A right bracket can be in the list only if it appears first. Thus, [A-Za-z0-9] matches an English letter or digit, and [][] matches a square bracket.

To print all symbols, use **who("*")**.

xd( ) — **xd( )** exchanges the top two entries in the current directory stack, making the new top entry the current working directory, and calls **dirs( )** to print the updated stack. **xd( )** raises an error if there are not at least two directories in the stack.

See also **help_cd( )**, **help_dirs( )**, **help_popd( )**, **help_pushd( )**, and **help_pwd( )**.

Y0(x) — **Y0(x)** returns the Bessel function of the second kind of order 0 of **x**, for **x >= 0**. This function is also called *Weber's function*.

Y1(x) — **Y1(x)** returns the Bessel function of the second kind of order 1 of **x**, for **x >= 0**. This function is also called *Weber's function*.

Yn(n,x) — **Yn(n,x)** returns the Bessel function of the second kind of integral order **n** of **x**, for **x >= 0**. This function is also called *Weber's function*.

ADDITIONAL LIBRARIES

hoc comes with a small collection of libraries that can be loaded with **load( )** commands, possibly placed in your personal startup file so that you always have selected libraries preloaded. Each function and procedure provided has a corresponding help procedure, so that documentation will not be repeated here.

annuity — Simple financial computations with functions **annuity( )** and **compound( )**, and procedure **mortgage( )**.

fortune — Numeric fortune cookies, with procedures **fortune( )** and **findfortune( )**.

primes — Prime number support, with functions **isprime( )**, **next_prime( )**, **nth_prime( )**, **prev_prime( )**, **this_or_next_prime( )**, and **this_or_prev_prime( )**, and procedures **prime_factors( )** and **primes_between( )**.

pushd — Procedures **dirs( )**, **popd( )**, **pushd( )**, and **xd( )**. [This library is useful enough that it is preloaded by default.]

require — Procedures **provide( )** and **require( )** for loading only libraries that have not already been loaded.

show-strftime Procedure **show_strftime_conversions( )** to test all of the format items provided by the **strftime( )** function.

sunmath Additional functions modeled on ones available in the Sun Solaris mathematical library: **exp10( )**, **exp2( )**, **iszero( )**, **max_normal( )**, **max_subnormal( )**, **min_normal( )**, **min_subnormal( )**, **quiet_nan( )**, **signaling_nan( )**, and **signbit( )**.

IMPLEMENTATION LIMITS

All internal storage areas in **hoc** grow as needed. There are no fixed limits on their size, other than the amount of available allocatable memory.

The current sizes of these internal storage areas are recorded as immutable numeric named constants:

__MAX_FRAME__	Function/procedure call stack size.
__MAX_LINE__	Input line buffer size.
__MAX_NAME__	Longest identifier name.
__MAX_PROG__	**hoc** virtual machine code size.
__MAX_PUSHBACK__	Input pushback buffer size.
__MAX_STACK__	Argument stack size.
__MAX_STRING__	Longest character string constant.
__MAX_TOKEN__	Longest numeric token.

This list may change during **hoc** development, but will ultimately be stable.

The function **help_limits()** can be conveniently used to display their current values.

EXAMPLES

```
func gcd(i,j) {
        ## gcd(i,j) returns the greatest common denominator of i and j
        temp = abs(i) % abs(j)
        if(temp == 0) return abs(j)
        return gcd(j, temp)
}

for(i=1; i<12; i++) print gcd(i,12)
print "\n"
1 2 3 4 1 6 1 4 3 2 1

### Print a table of the representable negative powers of 2
k = 0
x = 1
while (x > 0) \
{
        print "2(", k, ") = ", x, "\n"
        k--
        x /= 2
}
2^(0 ) = 1
2^(-1 ) = 0.5
2^(-2 ) = 0.25
2^(-3 ) = 0.125
. . .
2^(-1072 ) = 1.9762625833649862e-323
2^(-1073 ) = 9.8813129168249309e-324
2^(-1074 ) = 4.9406564584124654e-324
```

INITIALIZATION FILES

On startup, after processing any command-line options that suppress initialization files, **hoc** checks for the existence of local system-wide initialization files,

- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/hoc.rc`,
- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/locale/LN/hoc.rc`,
- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/help.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/locale/LN/help.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/translations.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.6.beta/locale/LN/translations.hoc`,

(LN is replaced by the locale name (see the **INTERNATIONALIZATION** section above), if one is defined, and otherwise, that file is omitted), and a private initialization file,

- `$HOME/.hocrc`,

in that order. Any that exist are automatically processed before the remaining command-line options are handled.

This feature allows for local customization of **hoc**, usually for additional constants and functions, as well as for locale-specific translations of output strings.

In initialization files, the **load( )**, **logfile( )**, and **save( )** commands are *always* available, even if command-line options disable them from use later in the job.

If GNU *readline* library support is available in **hoc**, then its initialization file, `$HOME/.inputrc`, (overriddable by setting an alternate filename in the value of the **INPUTRC** environment variable), can be used for customization of key bindings for command completion, editing, and recall. To restrict any such bindings to **hoc**, put them in a conditional like this:

```
$if hoc
    . . .
$endif
```

ENVIRONMENT VARIABLES

HOCPATH Colon-separated list of directories in which to search for input files. An empty component in the directory path list stands for the default **hoc** system path.

HOME User's home directory, where any private **hoc** startup file is stored.

INPUTRC Name of an alternate *readline* initialization file, overriding the default file, `$HOME/.inputrc`.

LC_ALL Primary variable defining the locale name. The name defines a component in the local installation's **hoc** library directory path in which are found localized files to support use of **hoc** in non-English environments.

LC_MESSAGES Secondary variable defining the locale name. It is ignored if **LC_ALL** is set.

LANG Tertiary variable defining the locale name. It is ignored if **LC_ALL** or **LC_MESSAGES** is set.

SEE ALSO

awk(1), **bc**(1), **dc**(1), **dircolors**(1), **emacs**(1), **expr**(1), **genius**(1), **locale**(1), **readline**(3), **vi**(1), **xlsfonts**(1), **xterm**(1).

FURTHER READING

This version of **hoc** grew out of the six generations presented in

Brian W. Kernighan and Rob Pike,
The UNIX Programming Environment
Prentice-Hall, Upper Saddle River, NJ (1984)
ISBN 0-13-937699-2 (hardcover), 0-13-937681-X (paperback),

LCCN: QA76.76.O63 K48 1984.

Sadly, most programming language textbooks have little or no coverage of floating-point arithmetic, and programming language standards, besides being hard to read, have generally provided inadequate support for IEEE 754 arithmetic.

An early draft of the IEEE 754 Standard was published in an October 1979 special issue of *ACM SIGNUM Newsletter*. The January 1980 and March 1981 issues of the IEEE journal *Computer* contain several papers about the then-developing IEEE 754 proposal, including a draft of the Standard.

The official IEEE 754 Standard is available as:
ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic
IEEE, New York, NY (1985)
20 pp.
ISBN 1-55937-653-8

Work on a revision of that Standard began about 2000, and is expected to take several years.

An interestingly account of the early development of the IEEE 754 arithmetic system can be found in the Web document
Charles Severance
An Interview with the Old Man of Floating-Point: Reminiscences elicited from William Kahan
URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`

The IEEE sponsors symposia on computer arithmetic that are held approximately every other year; the 15th was held in 2001. Most of the papers deal with low-level hardware issues of computer arithmetic.

The journal *Communications of the ACM* began publishing computer algorithms in 1960, and in 1974, that function was moved to a new journal, *ACM Transactions on Mathematical Software*. That journal, *TOMS* for short, has become the principal publication source for computer software that implements numerical algorithms. Other important journals in this area include *Computing*, *Mathematics of Computation*, and *Numerische Mathematik* (whose articles are mostly in English, despite the German title); their emphasis is often heavily theoretical.

A recent short book that discusses IEEE 754 arithmetic exclusively is:
Michael Overton
Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises
xiv + 104 pp.
SIAM, Philadelphia, PA (2001)
ISBN 0-89871-482-6
LCCN QA76.9.M35 O94 2001

Two recent books about hardware implementation of computer arithmetic are:
Amos R. Omondi
Computer Arithmetic Systems --- Algorithms, Architecture, Implementation
Prentice-Hall, Upper Saddle River, NJ (1994)
xvi + 520 pp.
ISBN 0-13-334301-4
LCCN QA76.9.C62 O46 1994

Behrooz Parhami
Computer Arithmetic: Algorithms and Hardware Designs
Oxford University Press, Oxford, UK (2000)
xx + 490 pp.
ISBN 0-19-512583-5
LCCN QA76.9.C62P37 1999

The older book
William J. Cody, Jr. and William Waite
Software Manual for the Elementary Functions

Prentice-Hall, Upper Saddle River, NJ (1980)
x + 269 pp.
ISBN 0-13-822064-6
LCCN QA331 .C635 1980

remains a good reference for the accurate computation of the elementary functions, and is one of the few to address the related issue of *decimal* floating-point systems (such as used in some hand calculators). Its elementary function test package, *ELEFUNT*, exposed serious flaws in a great many vendor implementations, and thanks to ELEFUNT, today, the accuracy and reliability of the revised implementations is very much better. Although the book was written before IEEE 754 arithmetic became available, in many cases, only simple tests for NaN and Infinity arguments need to be inserted into the start of each algorithm to generalize the code for current systems.

A excellent recent book that addresses computation of the elementary functions on a particular extended implementation of IEEE 754 arithmetic, that in the HP/Intel IA-64 architecture, is

Peter Markstein
IA-64 and Elementary Functions: Speed and Precision
xix + 298 pp.
Prentice-Hall, Upper Saddle River, NJ (2000)
ISBN 0-13-018348-2
LCCN QA76.9.A73 M365 2000

Markstein's book also contains algorithms for the correctly-rounded computation of floating-point division and square-root, and of integer division, starting from low-precision reciprocal approximations.

BUGS

All components of a **for** statement must be non-empty.

Error recovery is imperfect within function and procedure definitions.

The treatment of newlines is not exactly user-friendly.

Function/procedure arguments, whether named or numbered (`$1`, `$2`, ...) are not really variables and thus won't work in constructs like, for instance, `$1++`.

Functions and procedures typically have to be declared before use, which makes mutual recursion at first sight impossible. The workaround is to first define a dummy version of one of them. For example, here is an unusual implementation of a pair of functions, each of which returns the factorial of its argument:

```
func foo() return 0
func bar(n) {if (n > 0) return n * foo(n-1) else return 1}
func foo(n) {if (n > 0) return n * bar(n-1) else return 1}
```

AVAILABILITY

hoc is highly portable, and vastly smaller than a compiler for a major programming language, so it should be usable on all computing platforms. When a C or C++ compiler is available, **hoc** can be easily built, validated, and installed using the distribution source code from its master archive:

```
ftp://ftp.math.utah.edu/pub/hoc
http://www.math.utah.edu/pub/hoc/
```

For platforms where suitable compilers are often not installed, there may be binary distributions available at those locations.

COPYRIGHT

ACKNOWLEDGEMENTS

The **hoc** version 7 developer and maintainer (Nelson H. F. Beebe `<beebe@math.utah.edu>`) thanks the AT&T/Lucent Bell Labs people (current and former), notably Ken Thompson, Dennis Ritchie, Brian Kernighan, Rob Pike, John Bentley, Bill Plauger, Stu Feldman, David Gay, Norm Schryer, and Bjarne Stroustrup for developing the wonderful UNIX and C/C++ programming environment, for being a constant source of inspiration for software development, and for their superb book authoring.

He also thanks the many people at the Free Software Foundation, for enriching UNIX with GNUware, and most notably, Richard Stallman for **emacs**(1) and **gcc**(1), for founding the FSF and the GNU Project, and for vigorous campaigning to keep software freely distributable.

Finally, he thanks friends and colleagues on the **hoc** help facility translation team for assistance in internationalization: Hugo Bertete-Aguirre (Portuguese), Andrej Cherkaev (Russian), Tanya Damjanovic (Serbian), Michel Debar (French), Miguel Dumett (Spanish), Henryk Hecht (Polish), Michael Hohn (German), Ismail Küçük (Turkish), Young Seon Lee (Korean), Dragan Milicic (Croatian), and Jingyi Zhu (Chinese). [The English and Danish, and part of the French, help facilities were written by the maintainer.]